AD _____

TECHNICAL REPORT
GIT–ICS–81/12

LEVEL II

12

A REMOTE TERMINAL EMULATOR FOR PRIME COMPUTERS

17 34

By

Daniel H. Forsyth, Jr.

Prepared for

OFFICE OF NAVAL RESEARCH
800 N. QUINCY STREET
ARLINGTON, VIRGINIA 22217

Under

Contract No. N00014–79–C–0873
GIT Project No. G36–643

4100 44

August 1981

GEORGIA INSTITUTE OF TECHNOLOGY
A UNIT OF THE UNIVERSITY SYSTEM OF GEORGIA
SCHOOL OF INFORMATION AND COMPUTER SCIENCE
ATLANTA, GEORGIA 30332

1981

81 12 23 087

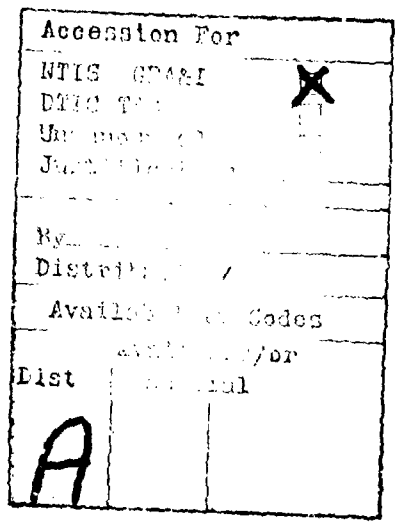
THE RESEARCH PROGRAM IN
FULLY DISTRIBUTED PROCESSING S

A REMOTE TERMINAL EMULATOR FOR PRIME COMPUTERS

TECHNICAL REPORT
GIT-ICS-81/12

Daniel H. Forsyth, Jr.

August, 1981



Office of Naval Research
800 N. Quincy Street
Arlington, Virginia 22217

Contract No. N00014-79-C-0873
GIT Project No. G36-643

The Georgia Tech Research Program in
Fully Distributed Processing Systems
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

THE VIEW, OPINIONS, AND/OR FINDINGS CONTAINED IN THIS REPORT ARE THOSE OF THE AUTHORS AND SHOULD NOT BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE NAVY POSITION, POLICY, OR DECISION, UNLESS SO DESIGNATED BY OTHER DOCUMENTATION.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: GIT-ICS-81/12	2. GOVT ACCESSION NO. AD-A108 818
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): A Remote Terminal Emulator For PRIME Computers	5. TYPE OF REPORT & PERIOD COVERED: Technical Report August 1981
	6. PERFORMING ORG. REPORT NUMBER: GIT-ICS-81/12 ✓
7. AUTHOR(s): Daniel H. Forsyth, Jr.	8. CONTRACT OR GRANT NUMBER(s): N00014-79-C-0873
9. PERFORMING ORGANIZATION NAME AND ADDRESS: School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, 800 N. Quincy Street, Arlington, Virginia 22217	12. REPORT DATE: August 1981
	13. NUMBER OF PAGES: 35 + viii
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: n/a

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution limited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

The view, opinions and/or findings contained in this report are those of the author and should not be construed as an official Department of the Navy position, policy, or decision unless so designated by other documentation.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Remote terminal emulator
Performance evaluation for time-shared systems
Benchmarking

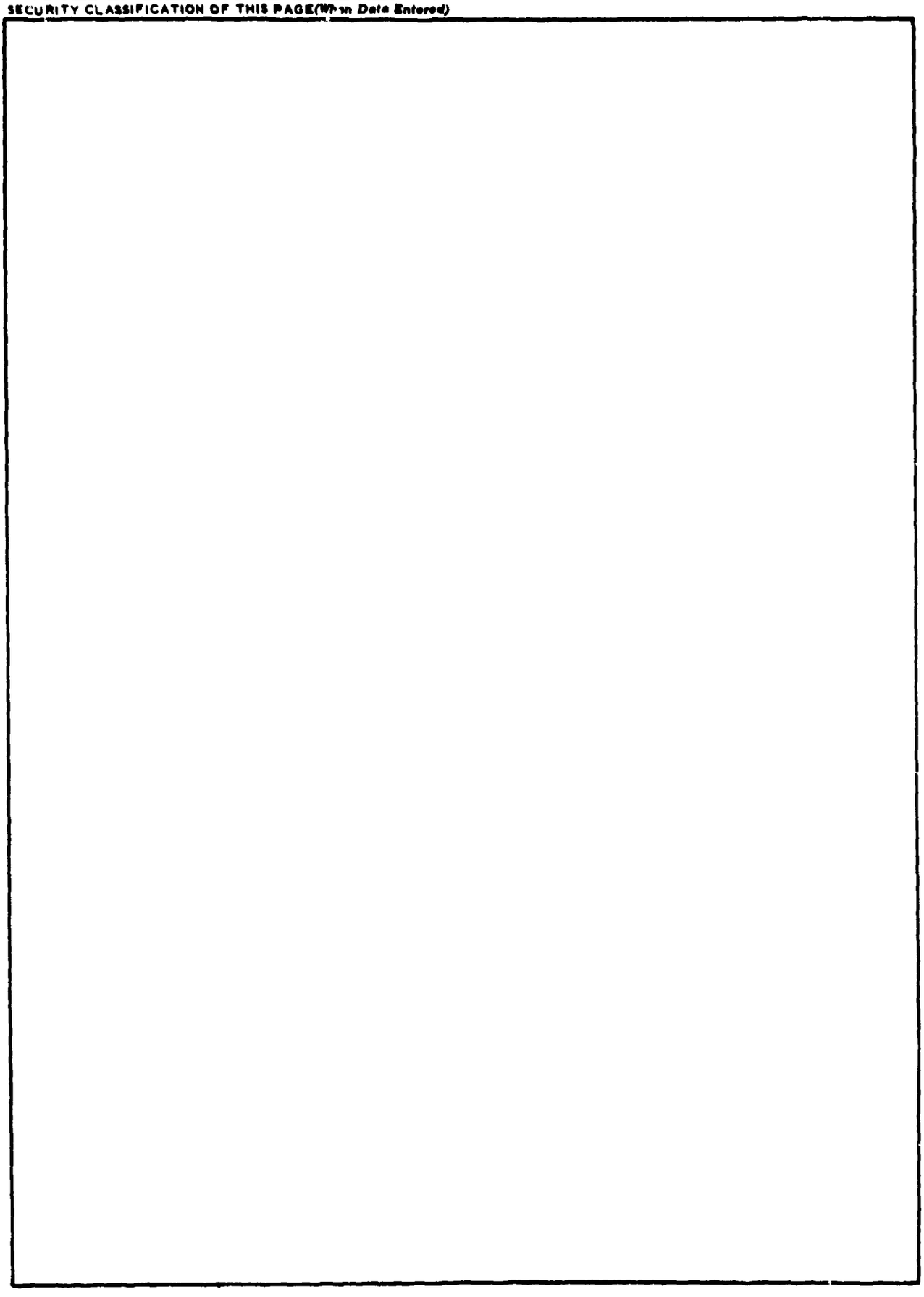
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A remote terminal emulator is a device that emulates sources of on-line input to a computer system, and is one of the most reliable tools for measuring the performance of time-shared computer systems. The remote terminal emulator described in this report provides simultaneous emulation for a number of communication lines as well as the software necessary for the preparation of scripts (a sequence of actions to be performed by the emulator) the setup and control of test sessions, and the analysis of test results.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)



Summary

A remote terminal emulator (RTE) is a device that emulates sources of on-line input to a computer system. An RTE is one of the most reliable tools for measuring the performance of time-shared computer systems. In order to emulate a wide variety of interactive input devices, an RTE is controlled by programs known as scripts. A script describes a sequence of actions to be performed by the RTE. Such a sequence might include messages to be transmitted to the system under test along with their timing, responses possible from the system under test, and actions to be taken after a specific response is received. As well as performing actions as specified by the scripts, the RTE records all the communication activity for later analysis.

The remote terminal emulator described herein runs on a Prime 400 or larger computer system under the standard vendor-supplied operating system. It provides simultaneous emulation for a number of communications lines as well as the software necessary for the preparation of scripts, the setup and control of test sessions, and the analysis of test results.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1

Introduction

> The performance, or operational behavior, of a software system is of prime importance to everyone concerned with the system — designers, implementors, and users. Obtaining and evaluating data on a system's performance must be an integral part of the process of creating systems [Freeman 75].

One of the goals of the on-going research in the School of Information and Computer Science is the creation of a testbed facility for the implementation and evaluation of fully distributed processing systems (FDPS). An essential feature of the testbed is the requirement to empirically evaluate the performance of fully distributed processing systems during their implementation. Providing a facility that measures these systems by generating an external load and measuring external response is the topic of this thesis.

1.1 Purposes of Performance Measurement

There are three general purposes of performance evaluation: selection evaluation, performance projection, and performance monitoring [Lucas 71]. These are shown in Figure 1.

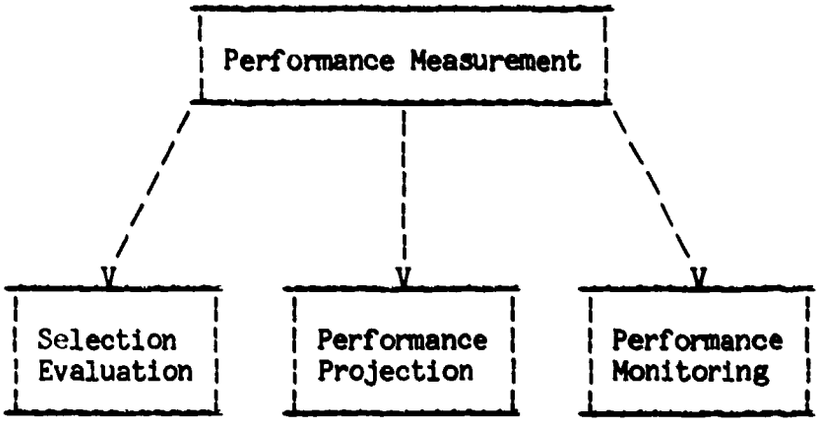


Figure 1 — Purposes of Performance Measurement

Selection evaluation involves the comparison of existing systems. The most frequent application of selection evaluation techniques is for comparison of computer systems to determine which system performs a given function most efficiently or whether a given system configuration can support a

particular application. Selection evaluation is also applicable when measuring the impact of different hardware or software on an existing system. For example, selection evaluation is useful in determining whether the addition of a load balancing algorithm improves interactive response time. Similarly, selection evaluation can answer the question "Did the last change to the operating system improve performance?" In all cases, the defining feature of a selection evaluation is that the systems to be compared must exist and must be available for testing.

Performance projection techniques are often applicable during the design of new hardware and software systems. These techniques attempt to predict the performance of new hardware and software designs prior to implementation. They can also be used to predict the performance of a system under a new workload or with a different hardware configuration. Performance projection techniques can often be applied to the same problems as selection evaluation techniques. However, the distinguishing feature is that it be not practical to actually test the systems under consideration: it may be too expensive to test the actual configuration, the configuration may not be available, or the system may not exist at all.

Performance monitoring techniques are applied in an attempt to understand the behavior of existing systems towards the goals of improving efficiency and service to users. It usually involves observing an existing system under normal operating conditions. Quantities measured with performance monitoring techniques are usually very dependent on the system measured (e. g., number of page faults, number of times the dispatcher is entered, etc.). For this reason, performance monitoring techniques are usually applicable only for the comparison of similarly structured existing systems. For instance, it is difficult to compare the performance of systems that use different disk block sizes by comparing the number of physical disk reads and writes.

In the ICS FDPS testbed facility, performance evaluation will be necessary for all three purposes. One need is for a performance measurement tool is in the area of selection evaluation. It is necessary to test prototype systems and compare the results with the results predicted by performance projection techniques, as well as with results obtained by testing other systems. The tool must be able to empirically measure the

performance of existing software and hardware configurations, and must be able to provide comparable measurements on similar configurations.

1.2 Techniques for Performance Measurement

A number of different performance measurement techniques can be applied for the purposes mentioned in the previous section. Figure 2 shows these techniques.

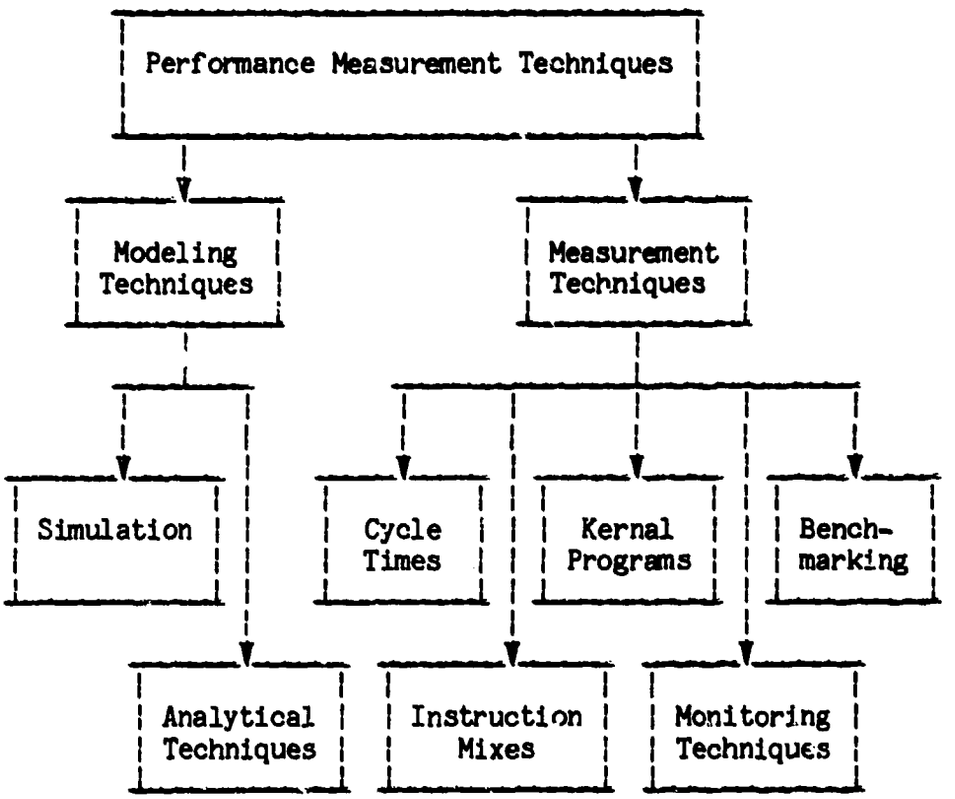


Figure 2 -- Techniques for Performance Measurement

Most of these techniques can be applied for all purposes of performance measurement, but some provide only marginally useful results. Since an FDPS testbed performance measurement tool is needed for the purpose of selection evaluation, the following discussion of performance measurement techniques is confined to those applicable to selection evaluation.

There are two classes of performance evaluation techniques that can be used for selection evaluation: modeling techniques and measurement techniques [Ferrari 78]. Modeling techniques involve building a

representation of the system to be evaluated and then testing that model. Although most useful in performance projection, modeling techniques can also be used for selection evaluation. A significant problem with all modeling techniques is determining how well the model reflects the system it models.

> Validation [of a model] is often difficult, and sometimes impossible. It may be based on previous theoretical or simulation results, but if the modeled system exists, the ultimate foundations of a validation model must be empirical. . . . Thus, in a sense, measurement is the most important evaluation technique, since it is needed also by the other techniques. [Ferrari 78]

Measurement techniques involve actually measuring the behavior of an existing system and thus are applicable only when the performance of a system can actually be determined. Several of the measurement techniques (instruction timings, instruction mixes, and kernel programs) merely make comparisons of hardware parameters such as memory cycle time, addition times, etc. These techniques are generally useful only as a supplement to more powerful techniques when used to compare hardware configurations and are inadequate when used to compare software systems [Lucas 71].

Hardware and software monitoring techniques, which usually involve the recording of such things as the number of page faults, number of cache misses, etc., provide a great deal of information about the performance of a particular system. But since the parameters that can be measured are usually very specific to a particular implementation, comparisons between systems with different internal structures are usually difficult to interpret.

The remaining measurement techniques, generally called benchmark techniques [Svobodova 76], involve actually running a system using a set of real or carefully contrived input and measuring the response of the system. Since the benchmark techniques treat the system under test as a "black box", measuring only stimuli and responses, they are immune to many of the problems of other measurement techniques. In general, the only significant difficulties of benchmark techniques are in the determination of the input to the system under test and in the analysis of the output of the system under test.

To support the FDPS testbed, the performance measurement must be capable of consistently applying arbitrary benchmarks to the the machines that are or will be a part of the testbed. It must also allow arbitrary analysis of the responses of the testbed equipment. This decision permits a generally useful tool for the testbed, while not encumbering or presupposing knowledge of the research issues of either the FDPS project or of benchmark techniques.

1.3 Benchmark Techniques

Performing a benchmark on a system first involves devising a workload to apply to the system under test. Svobodova defines the workload of a system as "the total of resource demands generated by the user community" [Svobodova 76]. Seen from the benchmark point of view, devising a workload is simply defining the set of inputs to be presented to the system under test. It is not a function of a remote terminal emulator to devise the workload to be used as the benchmark. The user must be responsible for devising a representative workload based on the system to be tested — the performance measurement tool need only be able to apply an arbitrary workload.

Once a set of benchmark jobs have been chosen and tested, the workload can be applied to a particular system configuration. A batch system may be tested by simply placing each job deck in the card reader at a preappointed time, and noting the time needed for the completion of all of the jobs. Testing a slightly different configuration presents no additional problems. The workload in this case is repeatable; it can be run several times on one system and barring malfunctions, one can expect similar results.

Testing of an interactive system is much more difficult. Since an interactive workload is generated by users entering data at terminals, it is very difficult to generate a repeatable workload without additional computer assistance. In general it is not possible to get a dozen or more people to type in commands in exactly the same order and "think" for exactly the same time for many consecutive test sessions. To obtain comparable results from several test sessions, it is necessary to have a means to emulates the actions of the interactive users and to repeat the same work-

load many times without tiring.

A Remote Terminal Emulator (hereafter referred to as an RTE) is just such a device. Its primary function is to emulate the load placed on a system by remote sources attached through communications links, such as terminals, sensors, and process controllers. RTEs are quite useful in performance measurement and evaluation, as well as for emulating devices in multi-dropped line protocols, monitoring communication line activity, and providing a host system for the testing of communications line protocols.

When used for performance evaluation, the RTE must produce a predefined workload while recording data about the responses of the system under test. To be capable of generating an interactive workload as well as a batch workload, an RTE must be able to accurately emulate people typing at interactive terminals. An interactive session, as opposed to a batch job, has three additional characteristics: (1) future input may be determined by current output, (2) there may be pauses before input messages corresponding to user "think time", and (3) there are pauses between input characters corresponding to user typing rate [Svobodova 76].

For the needs of the FDPS testbed, a remote terminal emulator is best choice for the performance measurement tool. As a minimum, the RTE must be able to generate interactive workloads to drive the existing hardware and software in the testbed. Preferably the RTE should be a general tool for performing benchmarks; it should be able to emulate any interactive device, either computer system or terminal, that hardware considerations allow it to replace.

1.4 Other Work

Most major computer vendors support RTEs for measuring the performance of their systems. RTEs are invaluable for helping to tune computer systems to squeeze the most performance per dollar, as well as for helping to convince prospective customers that a particular configuration will indeed do what the specifications says it will. Other groups, such as the U. S. Air Force and Tymshare also use RTEs for assistance in tuning and selecting computer systems [Watkins 77]. Most existing RTEs run either on inexpensive minicomputers (Air Force/MITRE DVM) or on the same family of systems they are built to test (Burroughs System/DCEM, IBM DB/DC Driver,

Univac CS1100). Some performance measurement tools run in a front end or peripheral processor attached to the system to be tested (Honeywell Datus, CDC BARTER) [Watkins 77]. These tools have functions similar to RTEs, but are not usually classified as such:

> There are implementations of workload drivers which reside either within the SUT [system under test] or in its communications front-end. . . . SUT resource dependency excludes these specific implementations from the classification RTE. In the interests of control and repeatability of testing, and of creating as near a duplication as possible of a specified workload and its effect on the SUT, the driver must be external to and independent of the SUT for the device to be an RTE [Watkins 77].

Most existing remote terminal emulators have a structure similar to that shown in Figure 3. Scripts, either in original or compiled form are accepted by the emulator and used to generate messages to be sent to the system under test. The responses of the system under test (either the entire response or the critical portions such as the first and last characters) are either time-stamped and logged in the session log files or are immediately reduced into statistics that are being collected. Depending on the implementation, the RTE may or may not examine responses from the system under test to modify its future actions. At the end of the test session, the desired measurements can be obtained from the RTE or by analyzing the session log files.

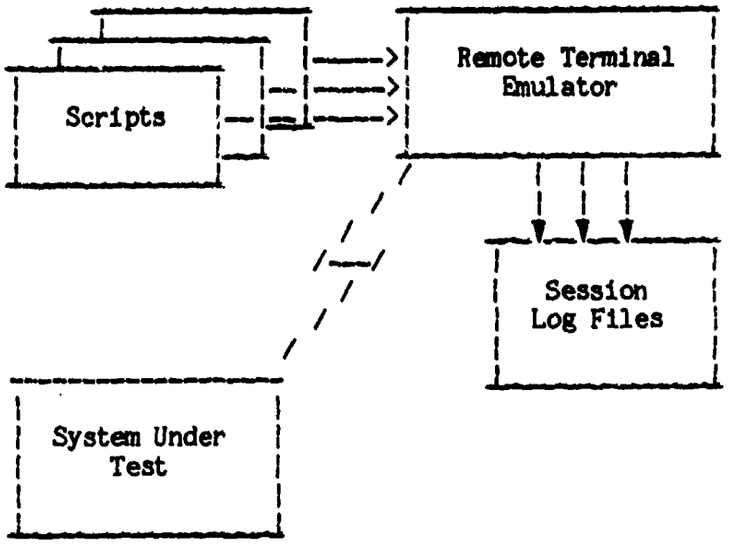


Figure 3 —— Remote Terminal Emulation

CHAPTER 2

The RTE Project

2.1 Objectives

From the preceding discussion of the motivations for the RTE project, two design objectives arise: the RTE should produce realistic interactive workloads and the RTE should remain an effective tool for several years. These objectives, although succinct, are not absolute requirements. It is necessary, as in most software projects, to compromise some of the objectives for practical reasons. For instance, extremely accurate time interval measurement cannot be provided without hardware modification. Requiring special hardware reduces the long-term usefuless of the RTE, but increasing its timing accuracy allows the generated workload to be more representative.

Two requirements are necessary to ensure the RTE's ability to generate realistic workloads: the RTE must be able to accurately emulate remote devices, and the workload presented by the RTE must be repeatable [Watkins 77]. These requirements are based on the primary motivation for the project: some method must be provided to accurately simulate real interactive users.

To be able to accurately emulate remote devices, the RTE must be capable of three things: it must be able to alter its behavior based on data it receives from the system under test, it must be able to accurately control delays between characters, and it must be able to accurately control delays between a response from the system under test and the next message from the RTE. These requirements follow directly from the defining characteristics of interactive workloads mentioned in the last chapter.

The necessity that the RTE produce a repeatable workload is a direct result of the purposes for which the RTE will be used. Since it will be used to compare different hardware and software configurations, it must be capable of generating the same workload time and again. This is not to say, however, that given the task of generating the same workload, the RTE will generate identical output. If the behavior of the system under test differs, of necessity, response of the RTE will differ. What must be expected is that "each time the RTE presents an activity to the SUT [system

under test] the observed performance differences are due to the SUT and not to the RTE" [Watkins 77].

The requirements to ensure the long-term effectiveness of the RTE are perhaps more obvious, since they apply to most software systems as well. These include ease of use, ease of maintenance, and flexibility. It is clear that implementation of the RTE will have been wasted if use of the RTE requires as much effort and knowledge as is required to implement a special program to be used once to perform the same actions.

The RTE will not be useful if it is not easy to maintain (e. g., it requires a non-standard environment with its own special operating system and dozens of control files). Again, it will be pointless to keep the RTE if it requires more effort to maintain than it does to implement the special purpose programs the RTE replaces.

Finally, although the RTE must be easy to use, it must be flexible enough to perform complex and varied emulation tasks. A priori restrictions must be avoided that prevent the RTE from performing such tasks as simulating interactive devices other than user terminals, generating workloads for machines other than those in the FDPS testbed, posing as one or several terminals on a multi-dropped communications line, passively monitoring activity on a communications line, or emulating a host system for testing communications line protocols. The RTE must also be efficient enough to provide a number of concurrent sessions. Otherwise, the RTE will be of little use in monitoring even the existing systems.

2.2 Design Decisions

In considering the objectives for the RTE and the hardware on which it is to be implemented, several alternatives for the design of the RTE are possible. Some of these alternatives can be immediately eliminated because they cannot possibly meet the requirements established for the implementation; in other cases, a choice must be made for less concrete reasons. In these cases, the choice has been made in favor of the simplest scheme, so that if it is found to be inadequate, it can be remedied at the least expense. The rest of this section describes the major design decisions and their rationale.

The first choice in selecting an implementation plan is the choice of operating systems. Here there are just three alternatives: the Prime-supplied single-user operating system (Primos II), the Prime-supplied multi-user operating system (Primos), and no operating system at all. After experience with design of a stand-along program for the Prime systems during a course taught in the winter quarter of 1981, it is obvious that totally abandoning the vendor-supplied operating systems would be an extremely expensive and time-consuming move, probably tripling the magnitude of the project. Therefore the only reasonable alternative is to select one of the vendor-supplied operating systems.

It is clear that the RTE must be able to support multiple concurrent interactive sessions, so some concurrency will be required in the RTE. The multi-user operating system supports multiple concurrent processes and virtual memory, while the single-user operating system does not. There are only two possible advantages in using the single-user operating system, assuming multiple processes are simulated to provide the necessary concurrency: code can be shared between processes, and process switching time can be minimized. These advantages are not significant though, since the multi-user operating system allows reentrant code to be shared between processes and, more importantly, makes use of the microcoded process exchange mechanism provided in the Prime systems, providing very fast, if not the fastest possible, process switching.

Since use of the single-user operating system provides no obvious benefits and because it would noticably complicate the project by requiring the implementation of process scheduling and concurrency primitives, use of the multi-user operating system is probably the best choice.

Choice of an implementation language for the RTE is suprisingly simple. At the time of implementation, only a few languages were available on the Prime systems: Cobol, Basic, Fortran, Ratfor, and assembly language. Assembly language might be the logical choice if ease of programming and maintenance was not considered. However, the assembly language for the Prime systems is quite complex and there are few people who program in it effectively. Cobol and Basic are probably not well suited for this type of programming; in addition, the Prime implementation of these languages is very slow. The choice then falls to either Fortran or Ratfor; since Ratfor

is a superset of Fortran and provides many features for writing easy-to-understand programs, it was the obvious choice.

Another area for choice is the structure of the RTE itself. There are three different structures that can be used for the RTE: the RTE can directly interpret a human-readable script during the emulation session, the RTE can compile a human-readable script into a machine language program, or the RTE can compile the human-readable script into an easy-to-interpret intermediate form for execution. The principle difficulty with the first choice is that it takes a great deal of time to parse a free-form program. Since the number of simultaneous interactive sessions that can be run may well be determined by CPU time requirements, it seems foolish to place the parsing load in the most time-critical area when better alternatives are available.

The second approach, compiling a script into machine language, solves the objection to the first approach by allowing a complex script language while allowing quick execution. It does, however, present two other problems. First, it does not allow the sharing of code between scripts (except between identical scripts), since each script would be a separate object program. Second, it would significantly complicate the implementation to directly generate machine code, and generating assembly language or Fortran would inconvenience users by requiring a great deal of time for compiling and linking the script programs.

The last approach, compiling scripts into an intermediate form, minimizes the deficiencies in both of the previous two approaches. It permits a complex source language, while permitting efficient interpretation. It also allows the interpreter code to be shared among the concurrent processes and is much easier to implement and maintain. It is this approach that was used.

A difficult area to address is the analysis to be done on the output from an RTE test session. Little is known about what information will be required in the analysis of a test session, since many of the projects that might use the RTE have not been devised. Because of this, it is necessary to defer the decisions on the exact kinds of analysis that can be performed. Fortunately, there is an approach which allows this quite simply. The RTE time-stamps and records all input and output from interactive ses-

sions during emulation. Insructions are written in the script to place various markers in this log along with the session transcription. Then, after the emulation session is complete, these logs can be analyzed. Since events of interest to the investigator have been tagged by markers in the log, time intervals can be easily computed, and other information can be derived as needed. This approach has the benefits that the analysis code is not built into the RTE and can thus be changed without danger to the integrity of the RTE code, and since a complete record of the emulation session is made, analyses may be run and rerun on the same session without the need of repeating the expensive emulation session.

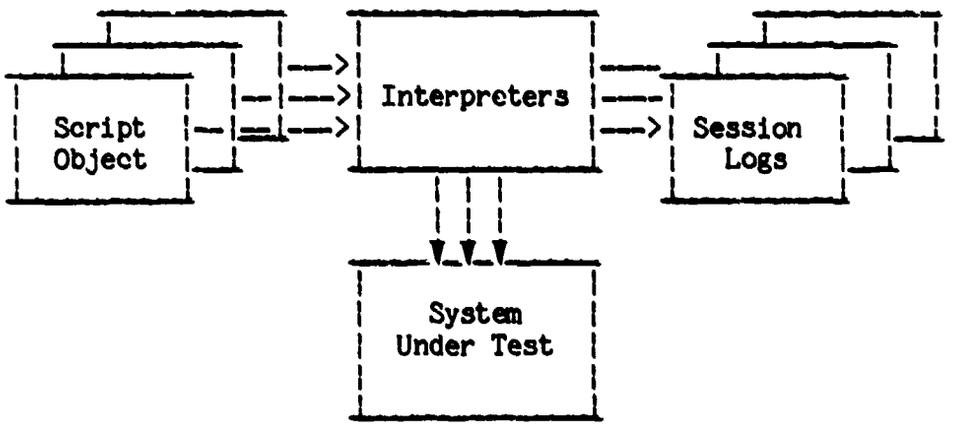
2.3 Implementation Sketch

The RTE is implemented on the FDPS testbed and runs under the Primos operating system on Prime 400 and larger systems with at least 1 megabyte of main memory. The code is written in a local dialect of Ratfor [Kernighan 76, Akin 80] which is part of the Georgia Tech Software Tools Subsystem [Akin 81].

As discussed above, RTE contains three components: the preprocessor, the interpreter, and the analyzer. A diagram of the structure of the RTE appears in Figure 4.

Preprocessor:

Interpreter:



Analyzer:

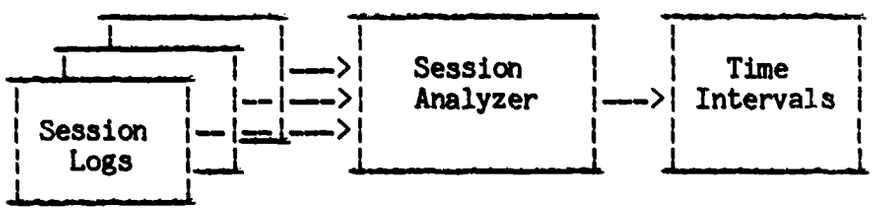


Figure 4 --- Structure of the RTE Implementation

Table 1 contains a brief description of how the RTE operates:

Table 1 --- Operation of the RTE

(1) The user first determines the scripts to be used. In addition to the text of the messages to be sent to the system under test, the scripts also contain information such as actions required for initialization, conditional branches based on responses from the system under test, and the method of computing character and message delay times.

(2) The user codes the scripts in the script source language and enters them with the text editor.

(3) The scripts are translated by the script preprocessor into the binary script object language.

(4) The user, with the assistance of the Software Tools Subsystem, creates a command procedure to initiate all instances of the interpreter. The user also specifies the script object program for each session, the method of access to the system under test, and the file to be used for logging session activity.

(5) The user initiates the command procedure. The interpreter processes are spawned, perform any necessary initialization and then wait for the signal to begin the emulation session.

(6) The user signals the interpreter processes to begin the test portion of their scripts.

(7) After the interpreter processes terminate, the user may then run the analysis programs using the session log files to obtain the desired measurments.

The implementation of the three components of the RTE are discussed below. Exact specifications and operating instructions for these components are described in the User's Manual, included as an appendix. Also described in detail in the User's Manual are the script source language, the script object language, the session log file format, and the analyzer output format.

2.3.1 The Script Preprocessor

The script preprocessor is responsible for converting user-written script source language programs into script object language programs for interpretation. The script source language is a simple line-independent language in which each statement begins with a keyword. Parsing is done with a simple recursive descent parser generated using the 'stacc' parser generator [Akin 81]. The preprocessor makes a single pass over the input language while generating two output streams. The first output stream contains directives for the allocation of all variables and constant data items. The second stream contains the code generated for each procedural statement. After the input has been completely processed, the preprocessor makes another pass over the procedure stream to backpatch the forward-chained branch addresses. Then the two streams are concatenated and copied into the script object file.

2.3.2 The Script Interpreter

The interpreter takes a single script object program and interprets it to emulate a single interactive session. Multiple sessions are emulated by running multiple copies of the interpreter simultaneously. When the interpreter is first executed, it reads the data allocation information from the object program and uses this information to initialize its data areas. It then begins to interpret the procedural code. In most instances, it performs any necessary initializations both locally and on the system under test (e. g., logging in, copy program and data files into a work area, etc.) By convention, the script directs the interpreter to wait for a signal from the user before beginning the test session code. When it receives the signal to continue (usually all sessions are signalled simultaneously), it begins to execute the test portion of the script program until that program directs it to terminate. Meanwhile, every input and output to the interpreter is time-stamped and logged for the analyzer. If the script program detects an error, or the interpreter receives no response from the system under test for a specified interval, it displays a message indicating the problem.

Currently, the interpreter has three ways of accessing a remote system. The first is through direct connection of an asynchronous line. This allows the interpreter to communicate with any device that supports an asynchronous RS-232-C interface. The second option allows the interpreter to obtain a virtual circuit to the system under test and initiate a terminal session through the remote login server of Primenet, thus allowing easy access to any Prime CPU connected via a ring network or other communications link. The third option causes the interpreter to read and write to its connected terminal, providing a way of exercising the interpreter and testing scripts without connecting to another computer (i.e. the user pretends that he is the remote system and types its intended responses). The interpreter is written so that all session establishment and input-output is performed in a single module so that as other access methods are needed, they can be easily added without concern for the structure of the rest of the interpreter.

2.3.3 The Session Analyzer

There are two kinds of analysis that can be performed on the output from an emulation session: session trace analysis and time interval analysis. The session trace analysis simply displays all input and output during the session in a readable form, along with the time intervals between each event. This type of analysis lends itself particularly well to verifying the actions performed by each script.

Time interval analysis displays the elapsed time between any two events occuring during emulation. Events are marked by "log" statements placed in the script by the user. For instance, to measure the time required to compile, link, and execute a Fortran program, the user would place a "log" statement for an event (perhaps event 1) just before the statement sending the "compile" command, and would place another "log" statement (perhaps event 2) just after the statement receiving the message indicating that execution is complete. The analysis program is then told to display the elapsed time between consecutive occurrences of event 1 and event 2.

All of the time interval analyses produce nothing more than a list of time intervals between the events that they measure. For instance, the measurement of the interval between completion of an RTE output and the receipt of the reponse message would result in the display of many time intervals (one for each output/response pair that occured), while the measurement of session time would result in the display of a single interval. These lists of raw time intervals may be easily concatenated and passed to any of several statistical packages, such as 'stats' [Akin 81] or SPSS [ref], to perform the desired statistical calculations.

CHAPTER 3

Conclusion

3.1 Evaluation

The requirements set forth for the RTE in Chapter II are summarized in Table 2:

Table 2 —— Requirements for an RTE

I. The RTE must be able to generate a useful workload
   A. accurately emulate remote devices
      1. alter its behavior based on data received
      2. accurately control delays between characters
      3. accurately control delays between messages
   B. present a statistically repeatable workload

II. The RTE must be useful in the long term
   A. easy to use
   B. easy to maintain
   C. flexible

The ensuing paragraphs discuss how well the RTE implementation meets each these requirements.

In choosing among the requirements, it is most important that the RTE be able to generate useful workloads. After examining the capabilities of the RTE implementation, it is clear that the RTE can accurately emulate remote devices: it can choose its execution path based on output from the system under test, it can be instructed to send a character after waiting an arbitrary time, and it can be instructed to wait an arbitrary period of time after recognizing a message from the system under test before transmitting the next input.

The only point of question is whether the implementation can accurately time the waiting periods and send the output characters at the time specified by the user. Since the timer resolution of the Prime CPU is 3.030 milliseconds, it is clear that one cannot expect the RTE to exceed this accuracy when depending on the system timer. In addition, character buffering and process scheduling mechanisms in the operating system have an incalculable, although substantial, effect on the accuracy of the timing.

When an interpreter process is awakened by the timer, several events must occur before it can send a character to the system under test. These events are outlined in table 3 below.

Table 3 —— Character Transmission Events

Interpreter process is notified. Timer resolution may introduce a delay error of up to 3 ms. in the awakening of a process scheduled to send a character.

Interpreter process is dispatched. A delay of several seconds may be introduced, depending on the operating system scheduler and CPU load.

The character is placed in the output buffer. Before reaching this point, the interpreter may be delayed even further by an unanticipated, asynchronous interrupt or it may be queued for having used up its current time slice.

The character is sent. Several milliseconds (unpredictable and uncontrollable) may elapse before the asynchronous line control hardware finds and sends the character.

Each of these time-dependent events delays the transmission of each character contributing to the timing error an increment ranging from a few milliseconds to several seconds.

The RTE attempts to avoid the cumulative error problem by sending characters at times specified relative to the beginning of the message, rather than specified as an interval between characters. This approach reduces the timing error between characters, but does not affect the timing error between messages.

To address the question as to whether the RTE can generate repeatable workloads, one must recognize that, although the behavior of each script should be deterministic, the overall result of a test session is non-deterministic. The probability distributions used for determining inter-character delays in the scripts are calculated by the script preprocessor and the exact values applicable to each specific delay are entered into the script object program by the preprocessor. Then the interpretation (i. e., execution) of a script should be completely deterministic. However, both the system under test and the RTE host system behave non-deterministicly during the test session.

One cannot (and should not) control the non-deterministic behavior of the system under test. As a consequence, it introduces a degree of randomness into the test results. The system under test may respond faster or slower to a given message and cause the corresponding interpreter process to execute faster or slower than its siblings.

The RTE host system, also a multiprogramming system, introduces another degree of randomness. As outlined in Table 3, there are several non-deterministic delays introduced on the timing of the output of each character. Although the variance of some of these delays can be controlled by manipulation of the operating system configuration parameters, most cannot be controlled without modification to the operating system algorithms. This is the penalty paid for the convenience of using the multi-user operating system in the RTE host.

To determine whether the RTE can generate repeatable workloads, one must answer the question about the accuracy of the timing. As discussed above, the script preprocessor calculates all probability distributions, so the script object program is deterministic. Since the interpreter can use the same script object program many times, only timing considerations are in doubt.

There is a significant difficulty in determining the impact of the timing inaccuracies since the only available measurement tool is another Prime CPU running the same operating system. Without the use of special hardware or software, the timing accuracy of the RTE can only be roughly estimated. Measurements made by executing a single script and recording the time each character was received by another system indicate that the average timing delay error for each character was less than 23 milliseconds per character with a standard deviation of 28. However, when the number of simultaneous sessions were increased to 20, the average delay error was about 100 milliseconds per character with a standard deviation of 130 when measured over a 5 minute period.

Because the behavior of the system under test is non-deterministic, the result of a single test session is just a sample of a random variable. Regardless of the behavior of the RTE, the measurements are neither exact nor perfectly repeatable, except in the stochastic sense. Because of this, much of the inaccuracy and non-deterministic behavior of the RTE can be

ignored when emulating terminal users allowing the results to be used with a reasonable degree of confidence.

However, these measurements and arguments do not rigorously verify the timing accuracy of the RTE. In addition, changes that further affect timing accuracy may occur with revisions to the operating system or hardware. For any application that depends greatly on the accuracy of timing, the user should verify the timing accuracy using the necessary special purpose hardware.

In evaluating the RTE implementation with respect to the requirements for long-term effectiveness, it is very difficult to make a quantitative judgement. Since perfection in meeting these requirements is probably not attainable, it is more useful to view them as design goals and examine the length to which these goals were persued in the RTE implementation.

Several features of the RTE should make it easier to use than comparable systems. First, the script preprocessor provides several common probability density functions for specifying character delays. Although some users may wish to specify the delay for each character, it is probably sufficient for most users to supply the parameters of a probability density function and allow the preprocessor to compute the delays.

Ease of use was also a principal considerations in the design of the script source language. The language allows column and line independent input and accepts a simple, English-like syntax with each statement beginning with a keyword. Source file inclusion and macro facilities are also present to minimize the effort to create the number of slightly different scripts necessary for a complex emulation session.

The fact that scripts are processed before the emulation session is begun produces another benefit: Any syntatic errors made in the script are diagnosed and can be corrected before the emulation session begins. This is in contrast to not encountering a syntax error until the interpreter processes the statement, perhaps not until near the end of a long emulation session.

Another of the features that make the RTE easy to use (at least for simple emulation sessions) is the access to the remote login server in Primenet. A user may specify that the RTE make virtual connections to the

system under test through Primenet, rather than require hard wire connections from asynchronous port to asynchronous port. Use of the Primenet remote login server places a different overhead on the system under test and so may disturb careful measurements. However, the disturbance is usually limited to less than 1 percent of the CPU capacity, so this form of connection provides a very convenient vehicle for gross performance measurements.

To be generally useful as a performance measurement tool in the FDPS testbed, the RTE must not be tied into making measurements of particular hardware and software configurations. It should be flexible enough to be able to emulate terminals attached not only to systems already part of the FDPS testbed, but to other systems, both present and future. The RTE was designed to avoid dependencies on the system to be tested and to allow easy repair of dependencies that are discovered. For instance, although the interpreter expects ASCII character input, the input and output routines are isolated and extensible so that new hardware or software interfaces can be accomodated with little difficulty. Similarly, the session establishment procedures are isolated so that new session protocols can be added.

The RTE avoids dependence on particular system prompting conventions by supporting generalized text patterns and not insisting on fixed order of inputs and outputs. The pattern matching capability, also borrowed from [Kernighan 76], allows the matching of arbitrary prompts by simply writing a regular expression. Although these particular routines are character oriented, a new set of routines may be easily substituted. Since the scripts allow an arbitrary flow of control, it is not necessary that the system under test prompt for each input, or even give the same prompt. The script can be programmed to recognize particular prompts from the system under test and choose its response accordingly.

Finally, ease of maintenance is a minimal requirement of any software system. In addition to adherance to good programming techniques, such as those suggested in [Kernighan 76], several considerations have been given to enhancing the maintainability and extensibility of the RTE. For instance, functions such as the statistical analysis of the time intervals produced by the analysis programs is left to existing packages, such as SPSS and 'stats'. Many of the other functions, such as setting up an

emulation session, have been left under the domain of the Software Tools Subsystem [Akin 81]. These systems are specialized to handle these particular functions; thus the RTE can consist of significantly less stand-alone code.

To further reduce the need for maintenance, the RTE implementation does not depend on local modifications to the operating system. It uses only documented, fully vendor-supported features. This allows the RTE to be used under the standard operating system releases; thus it is not necessary that a separate, specially modified operating system be maintained.

In the case that the existing script language is not powerful or fast enough to support an application, new capabilities are easily added with new statements to the script source language and new operators to the script object language. Instructions on how to accomplish this are part of the User's Guide that appears in the appendix.

Several utilities for displaying the script object language and the interpreter session logs are available for debugging both scripts and the RTE itself. These utilities understand these data structures and can print them in an easily readable form. Instructions for their use are also included in the User's Guide in the appendix.

3.2 Recommendations

There are a number of areas left to be addressed in the area of performance measurement tools for the FDPS testbed. Most important is research into how to effectively use the RTE. This thesis makes no attempt to suggest the content of scripts or the methods of analyzing test session results. These areas must be investigated by the user so that meaningful results are actually obtained by using the RTE.

Along the lines of more tractable projects, there are several areas in which the RTE can be improved. First, even though the script language is easy to use, it still requires a fair amount of work to create a script -- at least the same amount required to write a program of comparable complexity. A great help would be a program that spies on real users and generates scripts that perform exactly the same activities. At least then, a workload could be justified as "real", even if there is insufficient research to justify it as "representative".

Several parts of the RTE can also be improved. The interpreter must recognize that a significant time has passed so that it can report to the user that it has received no response. Currently, the interpreter must poll its input so that it is not blocked forever waiting on a character. A recent addition to the operating system allows a process to be signaled after a specified period of time, regardless of whether it is blocked waiting for input. Use of this feature would reduce the time necessary for busy waiting and perhaps allow more sessions to be run concurrently without suffering inaccuracy in timing.

The Software Tools pattern matching routines used for identifying responses from the system under test have two problems that should be approached: they are not particularly fast when matching patterns with closures, and they are inherently line oriented. Replacing them by a more sophisticated algorithm such as the DFA pattern matching algorithm described in [Aho 77] could significantly reduce processing requirements for complex patterns and remove any bias towards line separators.

ACKNOWLEDGEMENTS

I wish to thank my advisor, Dr. Philip H. Enslow, Jr., and the members of my committee, Drs. Richard J. LeBlanc and Nancy D. Griffeth, for their guidance and encouragement. I am especially grateful to Jeannette Myers for her gracious assistance in writing the programs and revising the thesis.

BIBLIOGRAPHY

Abrams 76 — Abrams, Marshall D. and Watkins, Shirley W., Summary of Findins on Alternatives to Remote Terminal Emulation for Imposition of Teleprocessing Workloads and Integrity Confirmation Aspects of Emulating Teleprocessing Workloads, General Services Administration, CS 77-4, November 1976.

Abrams 78 — Abrams, Marshall D., "Guidelines for the Measurement of Interactive Computer Service Response Time and Turnaround Time," Federal Information Processing Standards Publication 57, National Technical Information Service, August 1978.

Aho 77 — Aho, Alfred V. and Ullman, Jeffrey D., Principles of Compiler Design, Addison Wesley, 1977.

Akin 81 — Akin, T. Allen, Flinn, Perry B., and Forsyth, Daniel H., Georgia Tech Software Tools Subsystem Reference Manual, School of information and Computer Science, Georgia Institute of Technolgy, GIT-ICS-80/03, 1980.

Akin 80 — Akin, T. Allen, Flinn, Perry B., and Forsyth, Daniel H., Georgia Tech Software Tools Subsystem Reference Manual, School of Information and Computer Science, Georgia Institute of Technology, 1981.

DeMeis 69 — DeMeis, W. M., Weizer, N., "Measurement and Analysis of a Demand Paging Time-Sharing System," Proceedings of the 24th ACM National Conference, 1969, pp. 201-216.

Ferrari 70 — Ferrari, Domenico, "Architecture and Instrumentation in a Modular Interactive System," Computer, vol. 13, no. 8, August 1970, pp. 495-500.

Ferrari 72 — Ferrari, Domenico, "Workload Characterization and Selection in Computer Performance Measurement", Computer, July/August 1972, pp. 18-24.

Ferrari 78 — Ferrari, Domenico, Computer Systems Performance Evaluation, Prentice-Hall, 1978.

Freeman 75 — Freeman, Peter, Software Systems Principles, Science Research Associates, 1975.

Karush 69 — Karush, Arnold D., Two Approaches for Measuring the Performance of Time-Sharing Systems," Proceedings ACM - SIGOPS 2nd Symposium on Operating Systems Principles, Princeton University, October 1969, pp. 159-166.

Kernighan 76 — Kernighan, Brian W. and Plauger, P. J., Software Tools, Addison Wesley, 1976.

Lucas 71 — Lucas, Henry C., "Performance Evaluation and Monitoring", Computing Surveys, vol. 3, no. 3, 1971, pp. 79-91.

Nemeth 71 — Nemeth, Alan G. and Rovner, Paul D., "User Program Measurement in a Time-Shared Environment," Communications of the ACM, Vol.

14, No. 10, October 1971, pp. 661-666.

Nie 75 Nie, Norman H., et al., Statistical Package for the Social Sciences, McGraw-Hill, 2nd ed., 1975.

Rodriguez 77 Rodriguez, Humberto, Jr., Measuring User Characteristics on the Multics System, Laboratory for Computer Science, Massachusetts Institute of Technology, MIT/LCS/TM-89, May 1977.

Rodriguez-Rosell 72 Rodriguez-Rosell, Juan and Dupuy, Jean-Pierre, "The Evaluation of a Time-Sharing Page Demand System," AFIPS Proceedings of the SJCC, 1972, pp. 759-765.

Salzer 70 Salzer, Jerome H. and Gintell, John W., "The Instrumentation of Multics," Communications of the ACM, Vol. 13, No. 8, August 1970, pp. 495-500.

Scherr 66 Scherr, Allan S., "Time-Sharing Measurement," Datamation, Vol. 12, No. 4, April 1966, pp. 559-569.

Schwemm 72 Schwemm, Richard E., "Experience Gained in the Development and Use of TSS," AFIPS Proceedings of the SJCC, 1972, pp. 559-569.

Stimler 69 Stimler, S., "Some Criteria for Time-Sharing System Performance," Communications of the ACM, Vol. 12, No. 1, January 1969, pp. 47-53.

Stone 80 Stone, Harold S. et al., Introduction to Computer Architecture, Science Research Associates, 2nd ed., 1980.

Svobodova 76 Svobodova, Liba, Computer Performance Measurement and Evaluation Methods: Analysis and Applications, Elsevier North-Holland, Inc., 1976.

Watkins 77 Watkins, Shirley W. and Abrams, Marshall D., Survey of Remote Terminal Emulators, National Bureau of Standards, 500-4, April 1977.